NoSQL
Data Base Managing System

Pedro Guimarães

MongoDB Team's View | Problems

- Development speed: break 40 year relational paradigm;
- Scale: Adapted to new hardware - parallelism, multicore, cloud, etc.;
- Scalability issues, adapt to "Big Data";
- Complex data: object to object, polymorphism;

MongoDB Team's View | Scaling alternatives

Vertically	Horizontally
small box -> bigger box	small box -> more boxes
One problem or failure means whole system down	Failures have to be solved and the system keeps working, but may not be safe

MongoDB Team's View | Guiding concepts

"Como propor um novo modelo de dados útil e funcional diante dos paradigmas de programação modernos?"

chave/valor? Muito limitada.
xml? Não é intuitivo nem ágil o suficiente.
document-oriented, **não-relacional**?
↓
JSON - Java Script Object Notation

- Padronização RFC, garante portabilidade

MongoDB provê um console para gerenciamento e acesso direto aos dados, baseado em javascript (**mongo**).

A linguagem de queries do MongoDB é o próprio JSON.

JSON Datatypes:

1. **null**
2. **number**
3. **string**
4. **boolean**
5. **objects/documents**
6. **arrays**

Basic structure of the MongoDB

Relacional	MongoDB
Database	Database
Table	Collection
Row	Document (Object)

Dot Notation || Mongo Query Language

```
db.collection.method( {  paramaters(JSON) } );
```

CRUD | A sample document!

A seguinte query:

```
db.mycollection.insert( { " _id" : "Q33", "x":3, "y:"abc", "z": "[1,2]" } );
```

Produzirá o seguinte resultado na base de dados:

```
{
  _id: "Q33",
  x: 3,
  y: "abc",
  z: [1,2]
}
```

Para obter este objeto, seria necessária apenas a seguinte query:

```
db.mycollection.find( { " _id" : "Q33" } );
```

CRUD | A sample document!

Uma maneira possível de representar este objeto em um banco de dados relacional:

Tabela T:

P	x	y
Q33	3	"abc"

Tabela T_Z:

P	z
Q33	1
Q33	2

Para obter este objeto, seria necessária a seguinte query:

```
select * from T, T_Z where T.P = T_Z.P and P= "Q33";
```

ou

```
select * from T join T_Z on (T.P = T_Z.P) where P="Q33";
```

CRUD | Creating a document

```
db.users.insert (          ← collection
  {
    name: "sue",           ← field: value  }
    age: 26,               ← field: value  } document
    status: "A"            ← field: value  }
  }
)
```

Analogous SQL query:

```
INSERT INTO users                         ← table
            ( name, age, status )         ← columns
VALUES      ( "sue", 26, "A" )            ← values/row
```

CRUD | Updating a document

```
db.users.update(                          ← collection
   { age: { $gt: 18 } },                  ← update criteria
   { $set: { status: "A" } },             ← update action
   { multi: true }                        ← update option
)
```

Analogous SQL query:

```
UPDATE users            ← table
SET    status = 'A'     ← update action
WHERE  age > 18         ← update criteria
```

CRUD | Updates & moves

Update operations can increase the size of the document. If a document **outgrows its current allocated** record space, MongoDB must allocate a new space and move the document to this new location.

To reduce the number of **moves**, MongoDB includes a small amount of **extra space**, or padding, when allocating the record space. This padding reduces the likelihood that a slight increase in document size will cause the document to exceed its allocated record size.

*"Records and documents are **almost** the same thing,*
but records have some more space in the end,
*since **a record** actually **contains** the document!"*

Record:
[[document] + [free space (padding)]]

Obs.: Mongo has a 16MB per document limit. In order to store large files, mongo has "gridFS".

CRUD | Updates

The instructions:

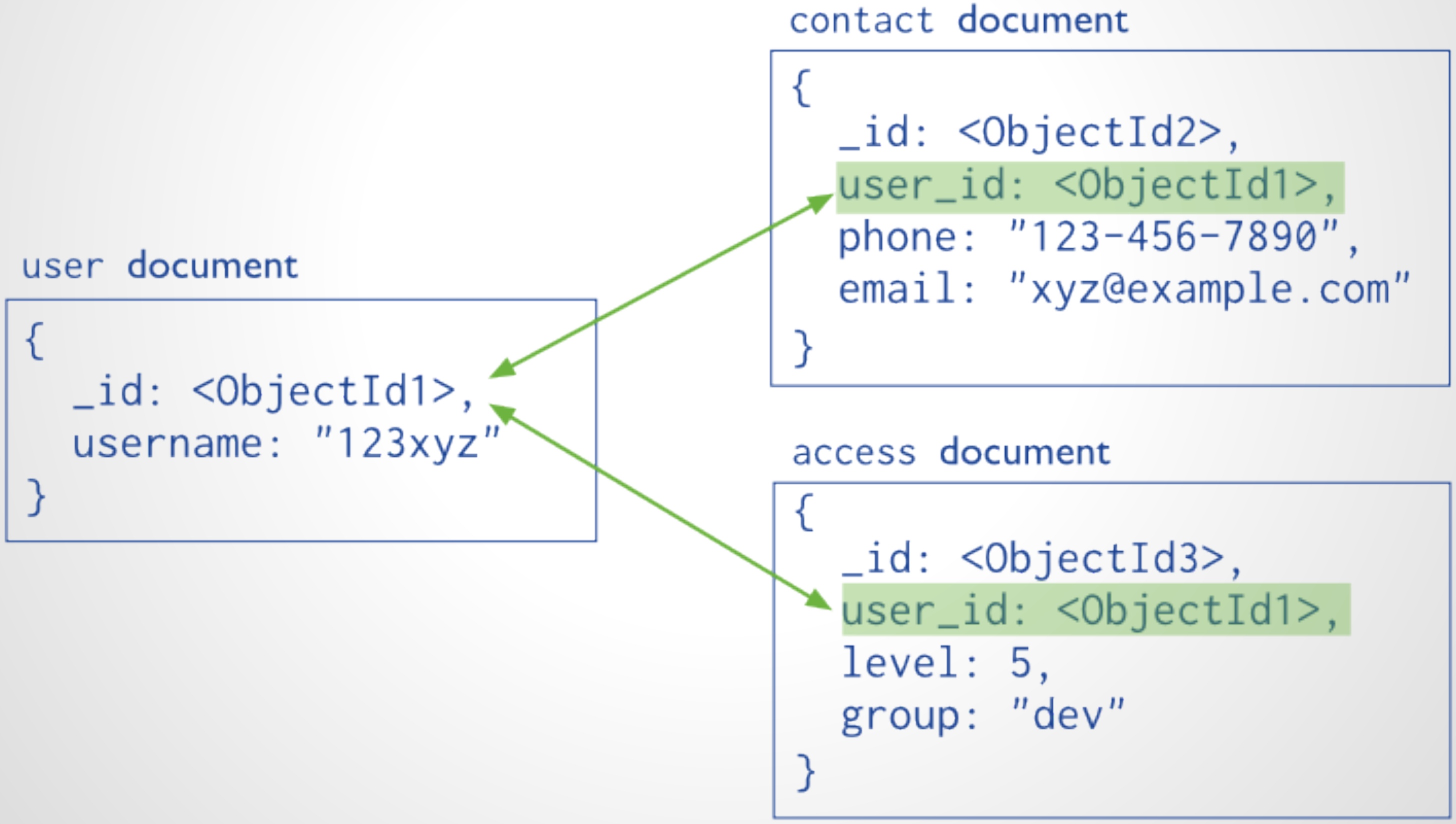
```
t = db.mycollection;
t.insert( { _id : 2, "z" : 17 } );
t.update ( {_id:2}, { $push: {"array":14} } )
t.update ( {_id:2}, { $push: {"array":"fifteen"} } )
t.update ( {_id:2}, { $push: {"array":"fifteen"} } )
t.update ( {_id:2}, { $addToSet: {"array":"16"} } )
t.update ( {_id:2}, { $addToSet: {"array":"16"} } )
t.update ( {_id:2}, { $addToSet: {"array":"16"} } )
```

Will result in:

```
{ "_id" : 2, "array" : [ 14, "fifteen", "fifteen", "16" ], "z" : 17 }
```

References

References offer a "normalized" structure for mongo documents.



Embedded data

Embedded documents capture relationships between data by storing related data in a single document structure.

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document (contact)

Embedded sub-document (access)

Indexing

MongoDB indexes use a **B-tree data structure.** In other words, indexes in Mongo DB are a **B-tree key (doc, location)**. One might define a key in either ascending or descending order, and it will serve both ways, since all the mongo algorithm has to do is choose whether to read the tree from **right to left**, or **left to right**.

- _id is implicitly called, all other indexes must be explicitly declared;
- Can index array contents;
- Can index subdocuments and subfields;
- Indexes may be of any kind (string, integer, etc).
- Allows multi-part indexes;

Types of cursors used by the query operations in indexing:

- BasicCursor indicates a **full collection scan**.
- BtreeCursor indicates that the query **used an index**. The cursor includes name of the index. When a query uses an index, the output of explain() includesindexBounds details.
- GeoSearchCursor indicates that the query used a **geospatial index**.

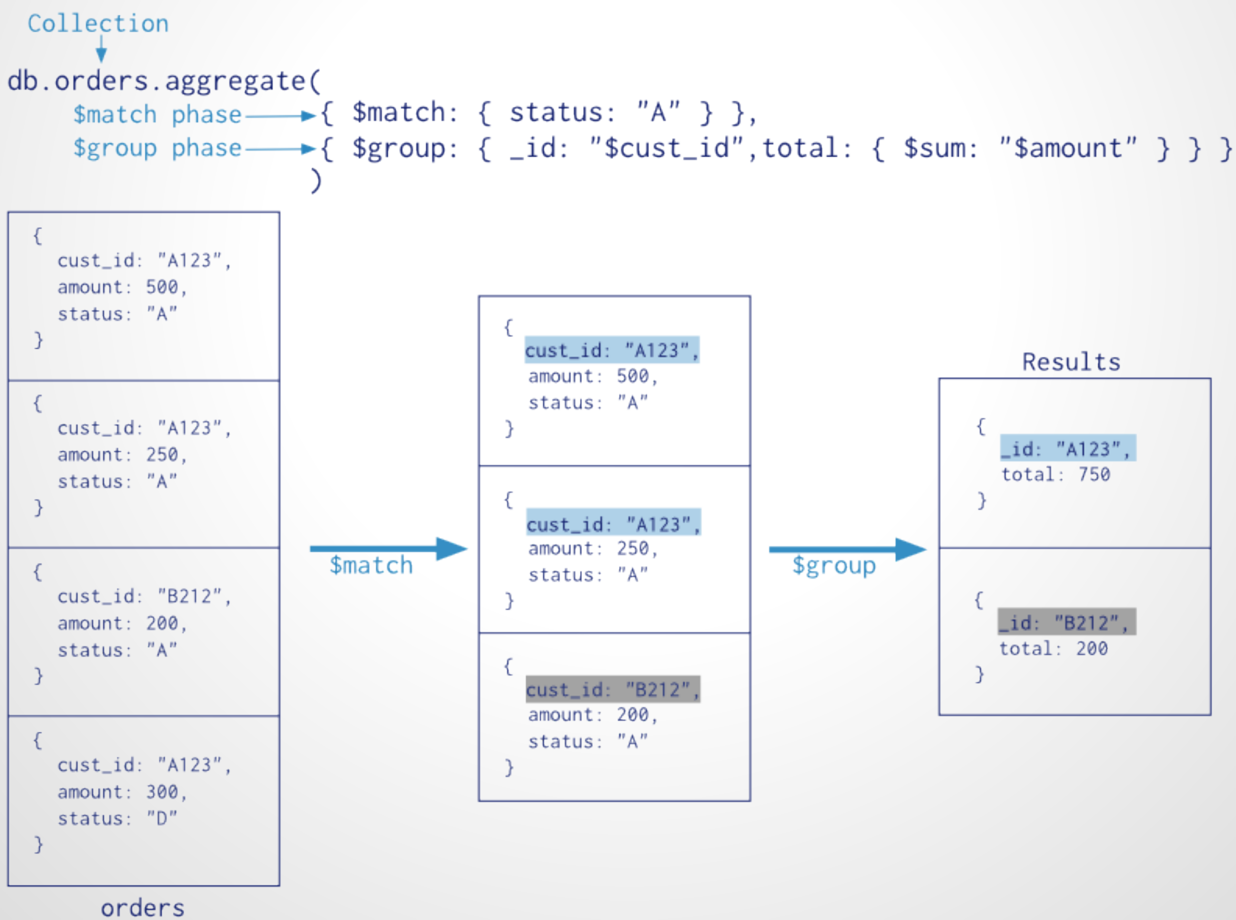
How to create an index:

```
db.collection.ensureIndex( { paramaters } );
```

Aggregation framework

Aggregation is a **multi-stage pipeline** that transforms the documents into an **aggregated result**, resembling somehow a **join in the relational world**. In a parallell with the pipe concept, it would feel like the following:

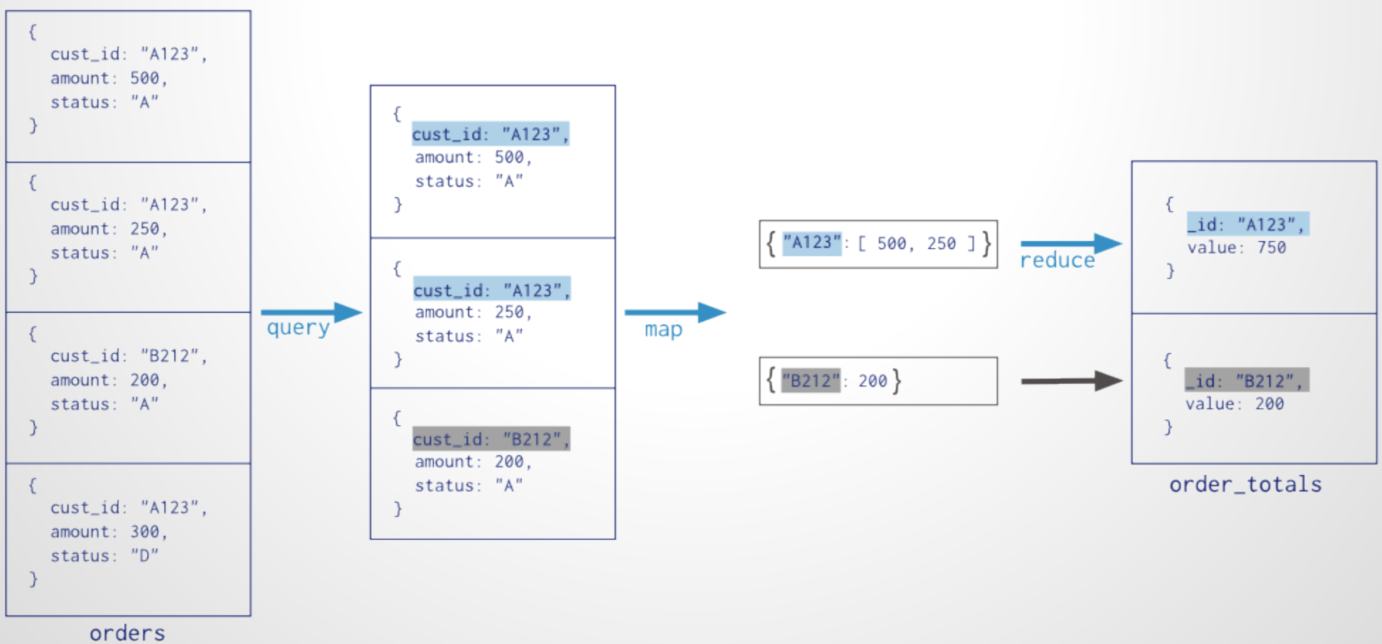
aggregate X | match | project | group | ...



Map/reduce

Map/Reduce is more powerful than aggregation operations. One can **also use the hadoop conector** already avaliable to run map/reduce operations in mongo.

```
Collection
    ↓
db.orders.mapReduce(
        map    →  function() { emit( this.cust_id, this.amount ); },
        reduce →  function(key, values) { return Array.sum( values ) },
                  {
        query  →    query: { status: "A" },
        output →    out: "order_totals"
                  }
)
```



Key features

Duas funcionalidades básicas do MongoDB:
Sharding & ReplicaSets

Spliting collections & Duplicating data

In different **sets**, you will have **same** documents.
In different **shards**, you will have **different** documents.

ReplicaSet:

1. Redundant copies of data;
2. Replicas, copies, backups;
3. Data safety (ds);
4. High avaliability (ha);
5. Disaster recovery (dr).

Shards:

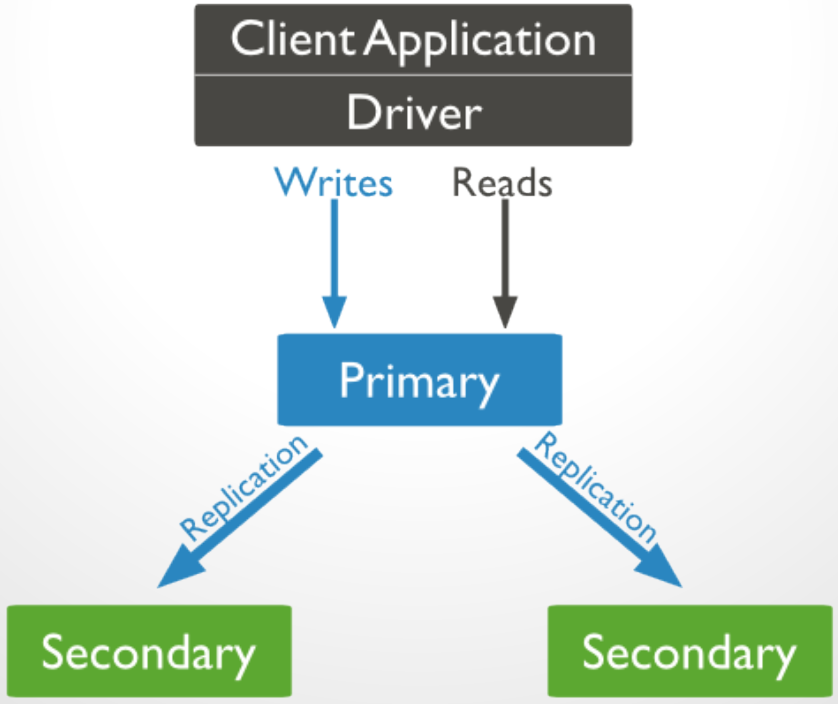
1. Unique data;
2. Scalability;
3. Data partition.

ReplicaSets

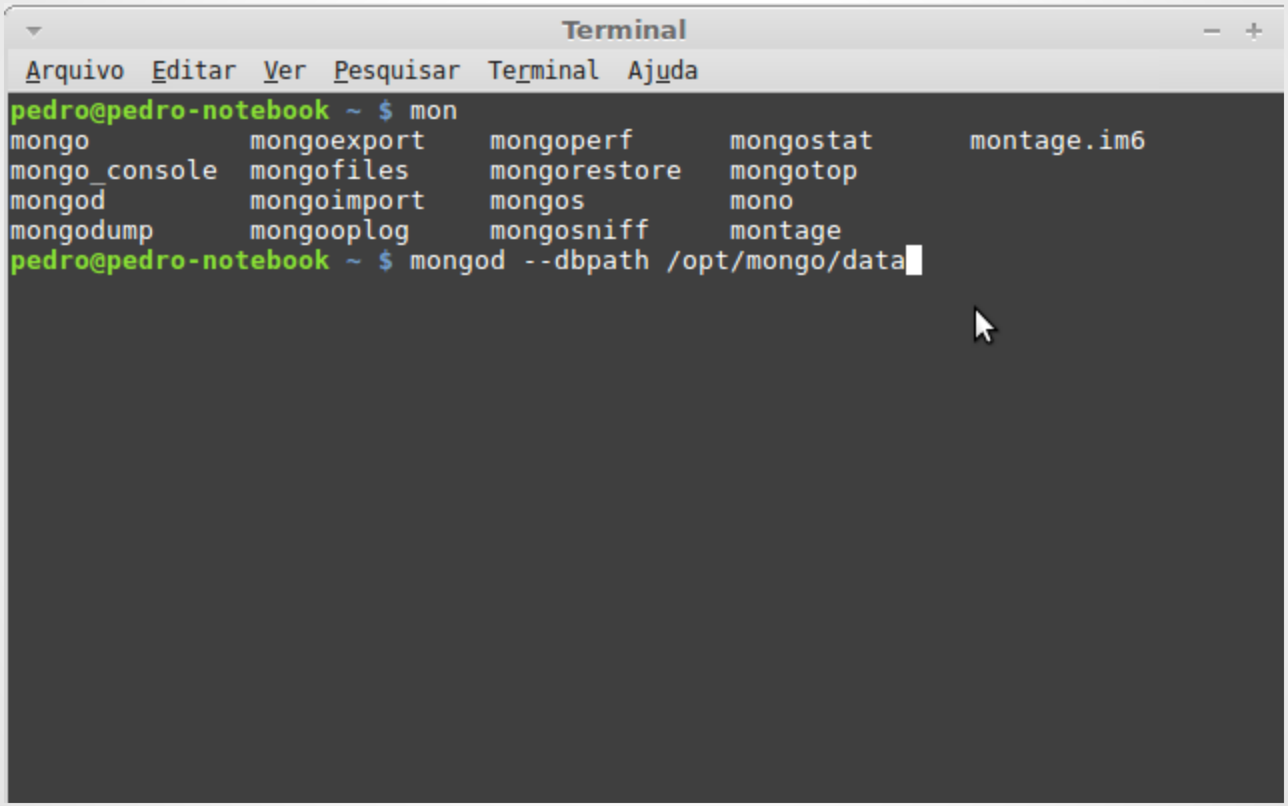
A **ReplicaSet** (rs) in MongoDB is a group of mongod processes that provide redundancy and high availability. The members of a replica set are:

Primary - receives all write operations

Secondary(ies) - replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be non-voting, have read prefence in reads, or be arbiters.



Showing how simple MongoDB is at the console!



Showing how simple MongoDB is at the console!

Terminal

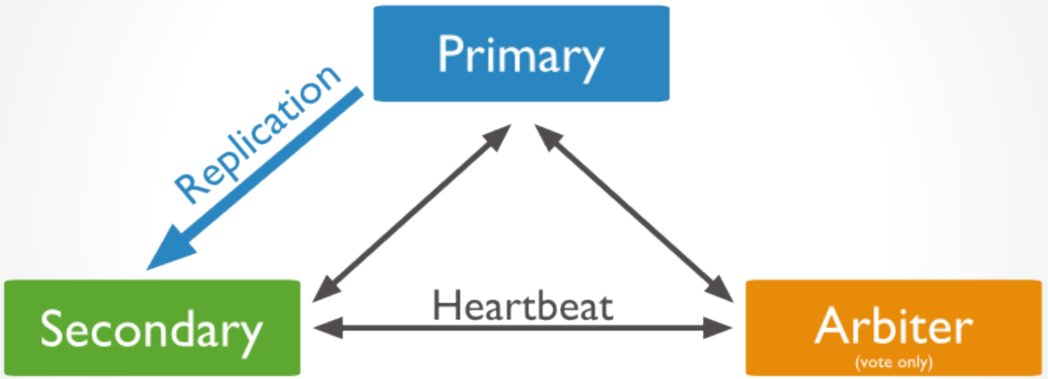
Arquivo Editar Ver Pesquisar Terminal Ajuda

```
pedro@pedro-notebook ~ $ mongo localhost
MongoDB shell version: 2.4.6
connecting to: localhost
> db.stats();
{
        "db" : "localhost",
        "collections" : 0,
        "objects" : 0,
        "avgObjSize" : 0,
        "dataSize" : 0,
        "storageSize" : 0,
        "numExtents" : 0,
        "indexes" : 0,
        "indexSize" : 0,
        "fileSize" : 0,
        "nsSizeMB" : 0,
        "dataFileVersion" : {

        },
        "ok" : 1
}
>
```

Arbiter

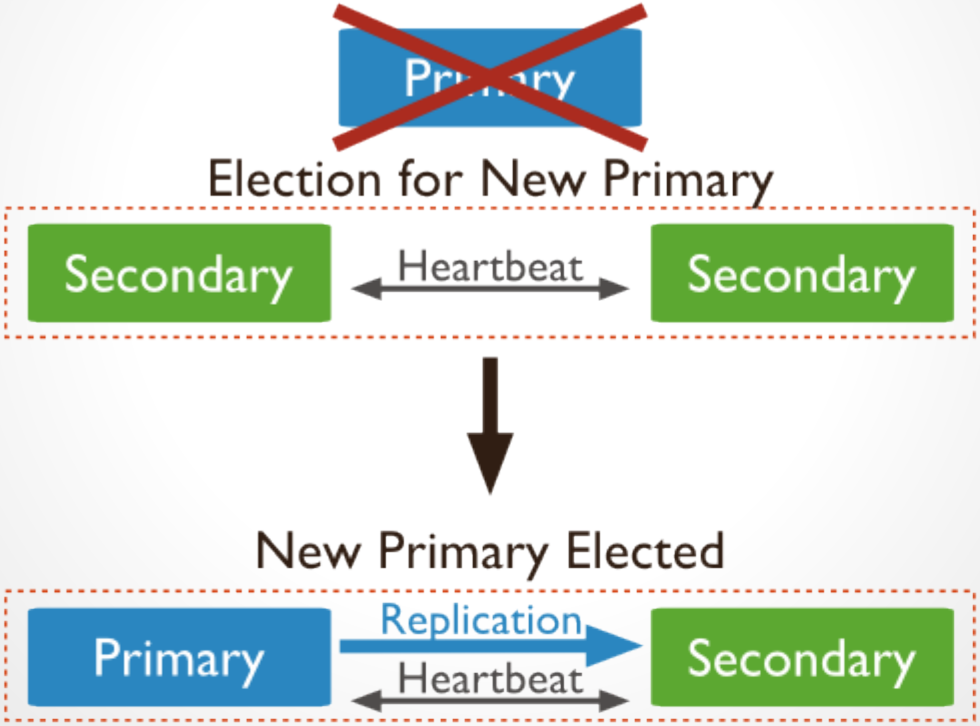
An arbiter **does not have a copy of data set** and **cannot become a primary.** Replica sets may have arbiters to add a vote in elections of for primary. Arbiters allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.



- Replicas have a priority level. Initially, all replicas have priority set as 1 (except arbiters).
- A member may have its priority set to 0, never being capable of being elected primary.
- Member with highest priority becomes primary.
- One member may have more than one vote.

Elections

Elections are essential for independent operation of a replica set. However, elections take time to complete. While an election is in process, the replica set **has no primary** and **cannot accept writes**. MongoDB avoids elections unless necessary.



Replication and Optime

Default replication is done **asynchronously** in MongoDB,
in concern for efficiency matters regarding distance issues in latency. Usually, the scenario is as follows:

Client →writes →Master/Primary →replicates →Slave/Secondary
↓
Client ← Acknowledges

Optime is a **tuple** used by mongod to register the write operations in the database, with two 32 bit fields:

<time>, <ordinal>

So, for instance, you might have something like:

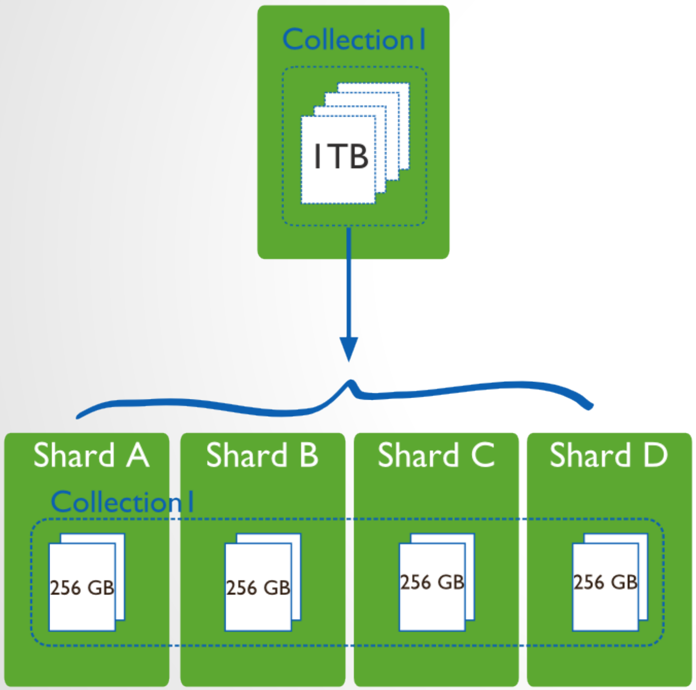
8nov2013 09:43:23 AM, 0
8nov2013 09:43:23 AM, 1
8nov2013 09:43:23 AM, 2
...
8nov2013 09:43:24 AM, 0
8nov2013 09:43:24 AM, 1
...

Obs.: since each field is a 32 bit value, mongoDB can not perform more than 4 billion operations per second!

So, knowing each server has its own optimes register, it is possible to know how big is the lag between mongod instances, both in terms of time and number of operations.

Sharding

Sharding **divides** the data set and **distributes** the data over multiple servers, or **shards**. Each shard is an **independent** database, and **collectively**, the shards make up a **single logical database**.

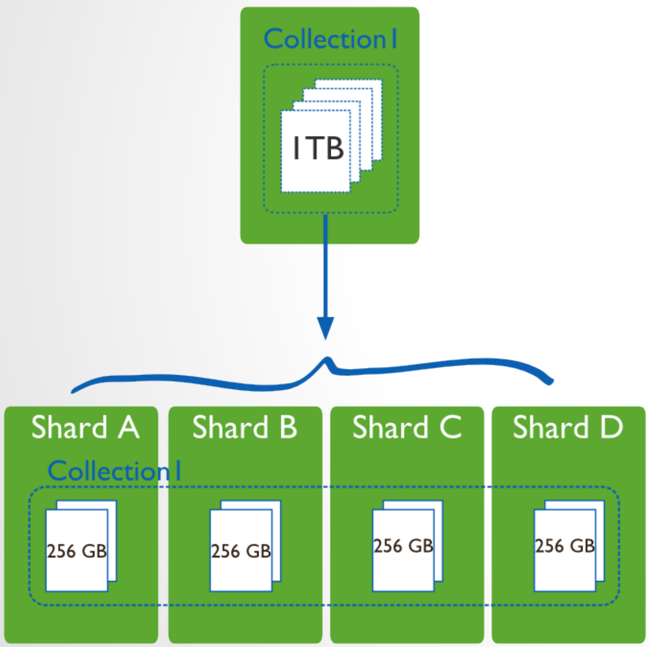


db.users

```
{
  _id: 1,
  name: "mario",
  likes: ["ski", "soccer", "swimming",
"judo"],
  age: 19
}
{
  _id: 2,
  name: sonia,
  likes: [basketball, tennis, dance]
  age: 24
}
...
{
  _id: 143565,
  name: albert,
  likes: [malakamb],
  age: 57
}
```

Sharding key

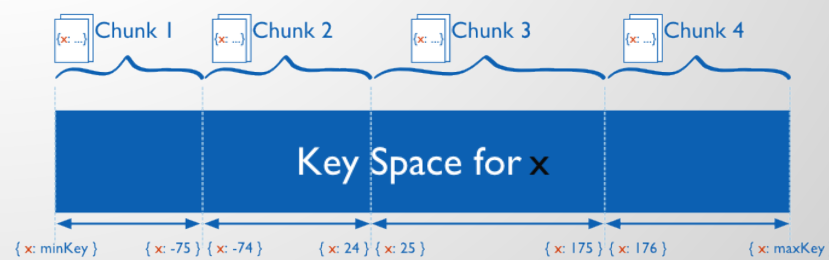
To shard a collection, you need to select a **shard key**. It is an **indexed field** or an **indexed compound field**. The shard key values are divided into **chunks** and **distributed evenly** across the shards. To divide the shard key values into chunks, MongoDB uses either **range based** partitioning (similar to Google's BigTable concept) or **hash based** partitioning.



One could think about sharding such collection in different ways. Either **_id**, **name** or **age** would be great candidates.

db.users

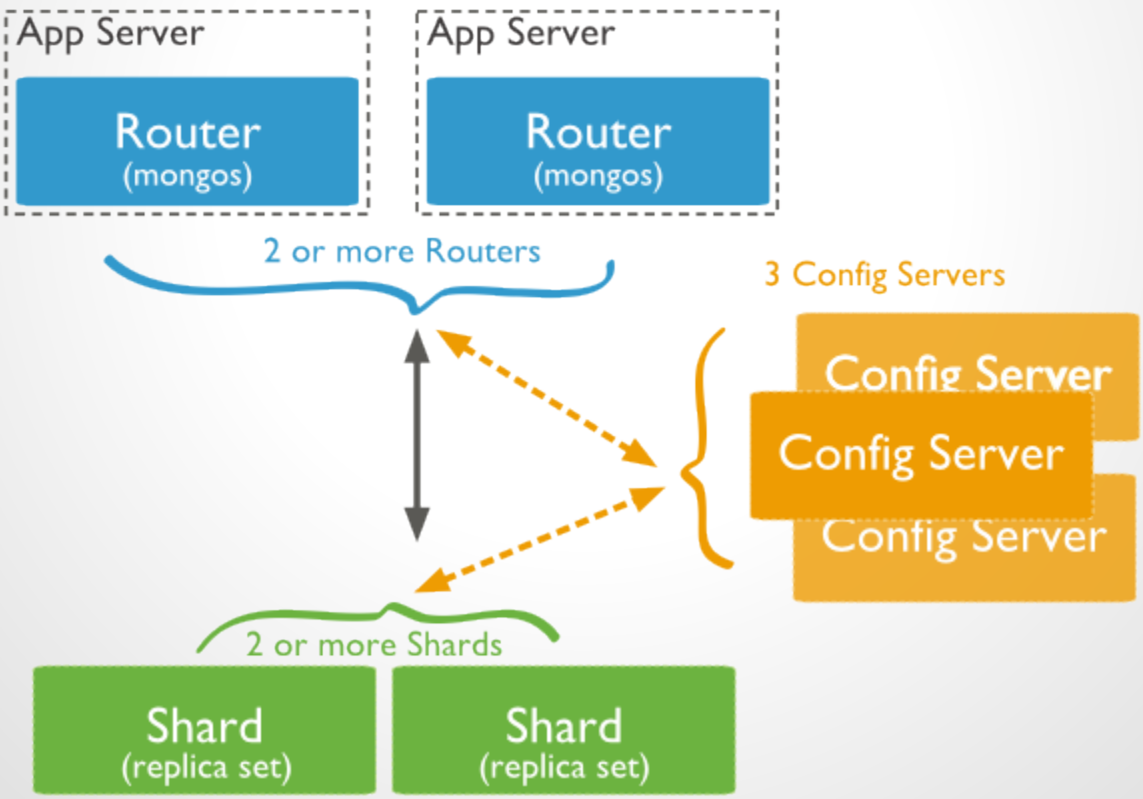
```
{
  _id: 1,
  name: "mario",
  likes: ["ski", "soccer", "swimming", "judo"],
  age: 19
}
{
  _id: 2,
  name: "sonia",
  likes: ["basketball", "tennis", "dance"]
  age: 24
}
...
{
  _id: 143565,
  name: "albert",
  likes: ["malakamb"],
  age: 57
}
```



Sharding + RaplicaSets!

In a realistic environment, we will have a mongo router filtering and centralizing accesses to the database. This process is a lightweight one called **mongos.**

mongos then will communicate to another process called **config server**, which is a light **mongod** process, without actual data, but with **metadata** about the entire server logical structure configuration. These servers send the requests to the appropriate mongod instance from the Shards/ReplicaSets.



Security

Possible strategies:

1) Mantain a "**trusted environment**", where you can lock down at the network layer all relevent tcp ports;

2) Use MongoDB **authentication**:

2.1) Using --auth for security client access, through user/password strategy;

2.2) Using --keyFile for intra-cluster security, granting that all servers that compose the grid are genuine and can be, therefore, trusted.

3) On top of that, one might **use SSL** to add encrypting to the messages exchanged within the cluster. But in order to do so, it is required to compile MongoDB especiffically for that, using the --ssl parameter.

Geospatial indexes

- 2D only
- Additional attribute ('compound')

Suppose you have a collection named "places" that looks like the following:

```
{
    _id: …,
    loc: [20.8, 43.1],
    type: 'coffee',
    …
}
```

To optimize accesses, one could add an index to this collection by doing: **db.places.ensureIndex({"loc": "2d"})**

And when you went for a query such as: **db.places.find({ loc: { $near: [20, 40], $maxDistance: 5 } })**

That would return all results within that range of a maximum distance of 5 between those measuring units.

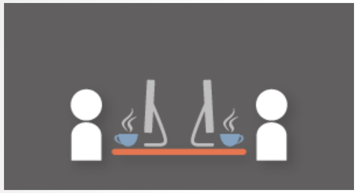
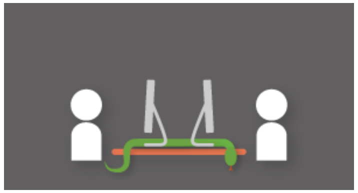
There is also a **geoNear command** implemented. That allows queries such as:

$within: { $center || $box || # $polygon }

Mongo also supports a **spherical: true** parameter to treat coordinates as spherical, for points at the surface of the Earth, for example.

MongoDB free online courses

Cursos: https://education.mongodb.com/courses

Cursos atualmente disponíveis:

1. MongoDB for Java Developers
2. MongoDB for Node.js Developers
3. MongoDB for Developers
4. MongoDB for DBAs

Gratidão!

Pedro Guimarães
pedrodpg@lncc.br